

Information Retrieval System Using Hadoop

Mayukh Sattiraju
Department of Electrical and
Computer Engineering,
Clemson
msattir@clemson.edu

Ashwin Parivallal
Department of
Mechanical Engineering
Clemson
apariva@clemson.edu

Abstract—Information Retrieval Systems are the core part in Search Engines. This document attempts to create a efficient retrieval system that, given a query can return documents that are most relevant to the query. In pursuit of this we discuss various search techniques, with increasing complexity and more efficient systems. We first introduce the problem and discuss the process we employed to tackle it. The first IR system we propose is a Boolean Search system. Then we propose a ranked retrieval system that uses tf-idf. Then finally we discuss a Probabilistic retrieval Model - the Okapi BM25 model to further improve the results. The report concludes with a comparison of these techniques and future work for the development of IR system.

I. INTRODUCTION

Big data is a term for data sets that are so large or complex that traditional data processing application software is inadequate to deal with them. Challenges include capture, storage, analysis, data curation, search, sharing, transfer, visualization, querying, updating and information privacy.

Information retrieval is the core part of any search related work such as search engines and database retrieval. Information retrieval (IR) is the activity of obtaining information resources relevant to an information need from a collection of information resources. Searches can be based on full-text or other content-based indexing. Information retrieval is the science of searching for information in a document, searching for document themselves, searching for metadata that describe data and for databases such as text, image or sound.

For our project we are going to use **Apache Hadoop** using Java as the API. Apache Hadoop is an open-source software framework used for distributed storage and processing of big data sets using the MapReduce programming model. It consists of computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common occurrences and should be automatically handled by the framework.

Map Reduce model - A MapReduce program is composed of a **Map()** procedure (method) that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

The Map Reduce model can be applied to many different problems and we will see how it helps us solve our problem in this project. It is mainly used on big data sets and the code is written in such a way that jobs are distributed evenly across all

the computers. In our project, we basically have two kinds of MapReduce jobs. One is for creating an inverted index for the Reuter News corpus and the other one is to create a Ranked and Boolean retrieval model to query on the inverted index created.

Reuters Corpus, Volume 2, Multilingual Corpus, 1996-08-20 to 1997-08-19. This is the corpus we will be using to create our database and query on the data. It has around a million files and each file is a news article written by local news reporters.

II. PROJECT DESCRIPTION

This is a semester long project divided into three phases. Each phase needs us to complete specific tasks to create our information retrieval system. Each phase is explained in detail in this report.

The first part of the project involves taking the **Reuters News data** and converting the required news articles words into an inverted index file. An **inverted index file** is basically a file containing information about all the words in the article and the information on documents that each term occur in. Taking this information as the basis, we construct our information retrieval system. The index file will be made with three flavours. **Uniword**, **Biword** and **Positional indices**. As the names suggest, our dictionary will contain single words in uniword index, two words in Biword and positional index will have a uniword term and will also have all the position of the terms in each document.

The second part of the project is to create different Ranked retrieval features. We have implemented a Boolean retrieval model, a ranked retrieval model based on tf-idf weights and a probabilistic retrieval model based on okapi BM25(Best Match).

Boolean retrieval model is what the name suggests. We can query the inverted index file using the basic Boolean operators like AND, OR and NOT. We can also have multiple operators in a single query. This can be easily computed by nesting the queries as needed. This kind of retrieval has its merits and also disadvantages. The advantages can be if the user exactly knows what he wants, then he/she can type it as a Boolean query. And this will give back exactly those documents that the user wanted. But on the downside, if the user is unsure of what exactly he is looking for, then this method may not give any results or may give absurd results.

Tf-IDF based retrieval is a ranked retrieval method that is most widely used in the industries. It gives very good results and ranks the documents based on the query the user entered. We have implemented this using the Map Reduce paradigm and it does a very good job at processing and computing the documents and rank it accordingly. This is a fairly good model for our information retrieval system. We will see further in the report that the weight on the documents logarithmically increases as the term frequency. This is justifiable since if the news article has many terms that are in the query, we can say that the news article is about the query.

BM25 probabilistic retrieval is supposedly the best among all the three models of retrieval. This model is based on weighting/ranking the documents based on the relevance with the query. More relevant the document is to the query, more weight is given to the document. The relevance can be measured in a couple of different ways. One may be based on the past views by the users and other may be based on cosine similarity between the document and the query. We have also implemented this model using the Map Reduce paradigm. So the final part of the project is the compare and contrast between these three methods of retrieval. Each method of retrieval can be good in certain cases and bad in others. These will be discussed at the end of the report.

III. PROJECT STAGE 1 - BOOLEAN RETRIEVAL

In this stage, we first create a database known as inverted index. This database consists of a dictionary having the following information:

- All the terms(words occurring in the news article)
- Document IDs for each term
- Term frequency over all the documents

The job starts with processing zip files one by one. For each zip file, all the XML files are extracted and the contents of the XML file is sent to the Mapper. This process is done using the ZipFileInputFormat file.

1) *Zip Format*: This is the java file that has the code for extracting the zip files and sending the contents of an XML file to the Mapper. It works based on the default FileInputFormat class provided by Hadoop and we extend this to process the zip files.

A. Mapper Phase

MAP INPUT: An XML files content

MAP OUTPUT: Term as key and its DocID as the value

Each Mapper takes in the contents of an XML file and parses it. The parsing of the XML file is done using the DOM parser of java. We mainly use the DocumentBuilder factory method to extract the content we want from the XML file.

The contents we retrieve from one XML file(one news article) are

- Document ID
- Headline
- Byline
- Text or the actual News content

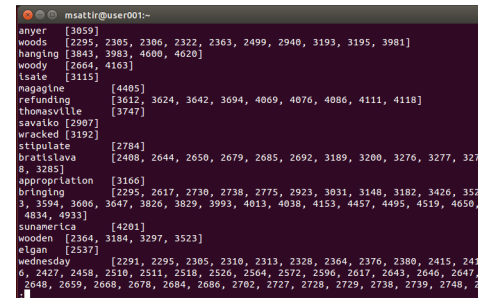
Once we retrieve the contents of an XML file, we process each word in the file as follows:

- For each word in the document, tokenize the word
- Stem the word (for example remove the plural)
- Remove any punctuation marks in the document

Once this is done, we send the word and its corresponding document id to the reducer.

B. Biword Mapper Process

The Mapper works only slightly differently for the creation of Biword inverted index. Instead of sending one term and its document ID, we send two adjacent term and its corresponding document id to the reducer.

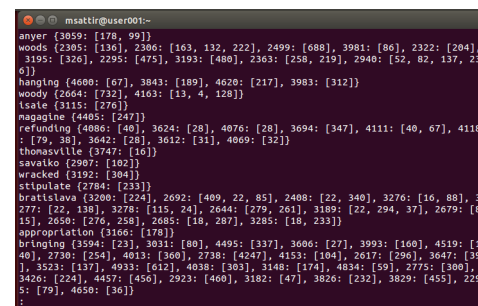


```
msattir@user001:~$ cat /dev/null
anyer [3059]
woods [2295, 2305, 2306, 2322, 2363, 2499, 2940, 3193, 3195, 3981]
hanging [3843, 3983, 4600, 4620]
woody [2664, 4163]
isate [3115]
nagazine [4405]
refunding [3612, 3624, 3642, 3694, 4069, 4076, 4086, 4111, 4118]
thonsville [3747]
savatko [2907]
wracked [3192]
stipulate [2784]
brattslava [2408, 2644, 2650, 2679, 2685, 2692, 3189, 3200, 3276, 3277, 3278, 3285]
appropriation [3166]
bringing [2295, 2617, 2730, 2738, 2775, 2923, 3031, 3148, 3182, 3426, 3521, 3594, 3606, 3647, 3826, 3829, 3993, 4013, 4038, 4153, 4457, 4495, 4519, 4650, 4834, 4933]
sunamerica [4201]
wooden [2364, 3184, 3297, 3523]
elgan [2537]
wednesday [2291, 2295, 2305, 2310, 2313, 2328, 2364, 2376, 2380, 2415, 2416, 2427, 2458, 2510, 2511, 2518, 2526, 2564, 2572, 2596, 2617, 2643, 2646, 2647, 2648, 2659, 2668, 2676, 2684, 2686, 2702, 2721, 2728, 2729, 2738, 2739, 2748, 2751]
```

Fig. 1. Inverted index for Uniword

C. Positional Index Mapper process

In this part, we will have to also store the position of each term occurring in the document. So we need to pass these positions to the reducer as well in addition to the term and its document id. The way we do this is by sending the position of the term along with the document ID. So, now the term will be the key of the output and Document ID and the position of the term relative to the start of the document will be sent as the value to the reducer.



```
msattir@user001:~$ cat /dev/null
anyer [3059: [170, 99]]
woods [2305: [136], 2306: [163, 132, 222], 2499: [688], 3981: [86], 2322: [204], 3195: [326], 2295: [475], 3193: [480], 2363: [258, 219], 2940: [52, 82, 137, 239]]
hanging [4600: [67], 3843: [189], 4620: [217], 3983: [312]]
woody [2664: [732], 4163: [13, 4, 128]]
isate [3115: [276]]
nagazine [4405: [247]]
refunding [4086: [40], 3624: [28], 4076: [28], 3694: [347], 4111: [40, 67], 4118: [79, 38], 3642: [28], 3612: [31], 4069: [32]]
thonsville [3747: [16]]
savatko [2907: [102]]
wracked [3192: [304]]
stipulate [2784: [233]]
brattslava [2408: [224], 2692: [409, 22, 85], 2408: [22, 340], 3276: [16, 88], 3277: [22, 130], 3278: [115, 24], 2644: [279, 261], 3189: [22, 294, 37], 2679: [15], 2650: [276, 250], 2685: [18, 287], 3285: [18, 233]]
appropriation [3166: [178]]
bringing [3594: [23], 3031: [80], 4495: [337], 3606: [27], 3993: [160], 4519: [140], 2730: [254], 4013: [360], 2738: [4247], 4153: [104], 2617: [296], 3647: [39], 3523: [137], 4933: [612], 4038: [303], 3148: [174], 4834: [59], 2775: [380], 3826: [224], 4457: [450], 2923: [400], 3182: [47], 3826: [232], 3829: [455], 2295: [79], 4650: [36]]
```

Fig. 2. Positional Inverted Index

D. Sort and shuffle phase

Sorting saves time for the reducer, helping it easily distinguish when a new reduce task should start. It simply starts a new reduce task, when the next key in the sorted input data is different than the previous, to put it simply. Each reduce task takes a list of key-value pairs, but it has to call the reduce()

method which takes a key-list(value) input, so it has to group values by key. It's easy to do so, if input data is pre-sorted (locally) in the map phase and simply merge-sorted in the reduce phase (since the reducers get data from many mappers).

Shuffling is the process of transferring data from the mappers to the reducers. Shuffling can start even before the map phase has finished, to save some time. That's why you can see a reduce status greater than 0

E. Combiner Phase

Combiner is a semi reducer phase in MR. Its an optional class which can be specified in MR driver class to process output of the map tasks before submitting it to the reducer.

In the MapRed framework, usually the output of the map tasks are large and the data transfer between the map and reducers will be high. Since the data transfer across the network is expensive and to limit the volume of data transfer between map and reduce tasks, we use combiners. Combine functions summarize the Map output with the same key and output of the combiner will be sent over the network to actual reduce task as input. Combiner doesn't have its own interface and it must implement reducer interface and the reduce method will be called for each map output key.

In our context, we have the Map output with term as the key and value as Document ID. The combiner will execute the reduce method and group all the documents with this term.

F. Reducer Phase

Reducer Input: Term as key and list of document IDs as values

Reducer output: Term and overall TF as key and array of Doc ids and TF for that document as values

The reducer phase doesn't have much to do. All it does is for each term, it stores all the documents it occurs in and also in this process it counts the number of times the term occurs. This process is repeated for all the terms in the corpus. Finally what we get out is the index file and this contains all the terms and its term frequency.

In case of the positional index creation, reducer has to do a little extra work. So in this section, the reducer will first see a term. For this term it will collect all the same document numbers and its positions. It will then group these positions and tag it along with the document ID. The same procedure is followed for all the documents in this term. Now, the same procedure is followed for all the terms.

G. Boolean Retrieval Process

The process of creating a Boolean retrieval is fairly straightforward once we have our index file. We accomplish this with MapReduce paradigm since the query will be processed faster if it's distributed.

H. Mapper Phase

Map input: The inverted index file **Map output:** Term as the key and its postings list as the value. The input will be the inverted index we created in the previous section. If we give the input path to the Mapper to be set to the index file,

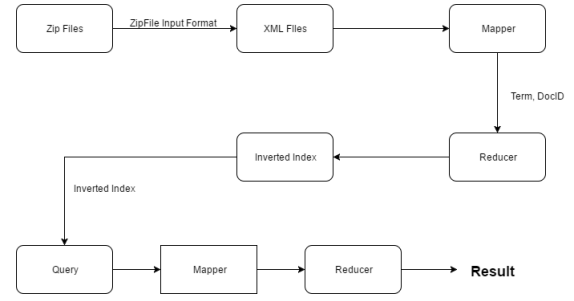


Fig. 3. Hadoop flow - Block Diagram

each term of the index file is given to a separate Mapper. So we have Mappers equal to the number of terms in the index file. This is just like doing a linear search on the index file. But here we are doing it in a distributed fashion and so the running time will be less than $O(n)$.

Note that we could have also created a custom FileInputFormat to split the index file into splits of specified length and give that split to a single Mapper. So this way we can decide the length of the input split and the number of terms that is processed by a single Mapper. Now we need to do a binary search on this split. This way we can get our results faster.

So as we execute each Mapper, we compare it with all the query terms, and if it matches with any of the query term, then we send that term and its postings list to the reducer.

I. Reducer Phase

Reducer input: term and the postings list **Reducer output:** List of documents that satisfies the Boolean query

In the reducer phase, we get all the terms from the query and the corresponding postings list. We now have all the terms and its postings list that are in the query terms. Now we execute the Boolean query based on the operator precedence. The precedence is in the order of NOT, AND and OR.

The way we do the Boolean query is to do it two terms at a time. We first execute NOT if needed.

The pseudo code for processing this is :

```

INTERSECT( $p_1, p_2$ )
1  answer  $\leftarrow \{ \}$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if docID( $p_1$ ) = docID( $p_2$ )
4     then ADD(answer, docID( $p_1$ ))
5      $p_1 \leftarrow \text{next}(p_1)$ 
6      $p_2 \leftarrow \text{next}(p_2)$ 
7  else if docID( $p_1$ ) < docID( $p_2$ )
8     then  $p_1 \leftarrow \text{next}(p_1)$ 
9     else  $p_2 \leftarrow \text{next}(p_2)$ 
10 return answer

```

Fig. 4. Pseudo Code - Boolean Search Reducer

IV. PROJECT STAGE 2 - RANKED RETRIEVAL

In this stage of the project, we were supposed to create a ranked retrieval method for retrieving documents from the inverted index file of Reuter news database. The inverted index file was created in the first stage of the project. But for this

stage we need to create a new index file with some extra information.

The three types of inverted indexes (uniword, biword , positional) remains the same. I have just added the ranked retrieval option in addition to the Boolean retrieval.

A. Ranked Retrieval Method

The most used ranked retrieval method in the industries is the tf-idf weights. I have also implemented the same. First let us have a little background on how this method works. There are two main components to this method namely, term frequency and inverse document frequency.

B. Term Frequency(tf)

Suppose we have a set of English text documents and wish to determine which document is most relevant to the query "the brown cow". A simple way to start out is by eliminating documents that do not contain all three words "the", "brown", and "cow", but this still leaves many documents. To further distinguish them, we might count the number of times each term occurs in each document; the number of times a term occurs in a document is called its term frequency. However, in the case where the length of documents vary greatly, adjustments are often made.

In the case of the term frequency $tf(t,d)$, the simplest choice is to use the raw count of a term in a document, i.e. the number of times that term t occurs in document d . If we denote the raw count by $f_{t,d}$, then the simplest tf scheme is $tf(t,d) = f_{t,d}$

$$tf(t,d) = \log(1 + f_{t,d})$$

Here, tf is zero if Frequency of Term $f_{t,d}$ is zero.

So the term frequency of a term t is the number of times it occurs in a document.

C. Inverse Document Frequency

Because the term "the" is so common, term frequency will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms "brown" and "cow". The term "the" is not a good keyword to distinguish relevant and non-relevant documents and terms, unlike the less common words "brown" and "cow". Hence an inverse document frequency factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

The inverse document frequency is a measure of how much information the word provides, that is, whether the term is common or rare across all documents. It is the logarithmically scaled inverse fraction of the documents that contain the word, obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient.

$$idf = \log\left(\frac{N}{n_t}\right)$$

Where N is the total number of documents in the corpus and n_t is the number of documents the term t occurs in.

D. Combined Ranking System

The tf-idf weight of a term is the product of its tf weight and the idf weight.

$$tf - idf_{t,d,D} = tf_{t,d} * idf_{t,D}$$

So the score for a document for a given query is the summation of the tf-idf weights of each term in the query. This is given by

$$Score(q,d) = \sum_t tf * id$$

E. TF-IDF Pseudo Code

COSINESCORE(q)

```

1 float Scores[N] = 0
2 float Length[N]
3 for each query term t
4 do calculate  $w_{t,q}$  and fetch postings list for t
5   for each pair( $d, tf_{t,d}$ ) in postings list
6     do Scores[d] +=  $w_{t,d} \times w_{t,q}$ 
7 Read the array Length
8 for each d
9 do Scores[d] = Scores[d] / Length[d]
10 return Top K components of Scores[]

```

Fig. 5. Pseudo Code - TF-IDF

F. Implementation of Map-Reduce Job

The reducer task is the main part of calculating the tf-idf score.

We first need to make an inverted index file for the whole Reuter corpus. Our Inverted index will have the format like:

What the above format means is that for each term t , we have a term frequency that indicates the total number of times this term appears in the documents. Then we have an array that has document id and the corresponding term frequency for that document.

G. Mapper Phase

Map input: The inverted index file

Map output: List of documents based on tf-idf weights

H. Calculating TF Weight

Using the index file we have the term frequency of all the terms in each document. So for each query term, we need to sum the tf weight in each document it occurs in. Similarly, we also need to calculate the inverse document frequency weight for each term in the query. We do this by taking the log of ratio of total number of documents in the corpus to the document frequency of that term. Then we just multiply the two weights.

V. PROJECT STAGE III - PROBABILISTIC RANKING

In information retrieval, Okapi BM25 (BM stands for Best Matching) is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. It is based on the probabilistic retrieval framework developed in the 1970s and 1980s by Stephen E. Robertson, Karen Sprck Jones, and others. The name of the actual ranking function is BM25.

To set the right context, however, it usually referred to as "Okapi BM25", since the Okapi information retrieval system, implemented at London's City University in the 1980s and 1990s, was the first system to implement this function.

We need to implement the ranked retrieval system using BM25 algorithm. The formula used for computing the weights of the documents is

$$\text{score}(D, Q) = \sum_{i=1}^n \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)},$$

Fig. 6. Okapi BM25 Formula

Where k and b are the parameters we can tune to get better results based on the types of query the user enters. For example, for longer queries the value of k should be on the lesser side [$k \in [1, 1.2]$]. For shorter queries its advisable to keep the value of k fairly high [$k \in [1, 1.2]$].

1) Parameters:

- The value of k determines how much importance we want to give to the term frequency. Keeping a low value of k will have a lower impact of tf on the BM25 weight for this document and vice versa.
- The value of b determines how much importance we want to give to the length of the document based on the query. If the length of the document under consideration is less than the average document length, then the tf score for this document is higher than the documents that are longer than the average document length.

A. Inverted Index

As we process the XML files, we need to pass additional information as compared to the tf -idf method. We need the terms and document IDs as before, in addition to that we also need to pass in the length of the document with each term. This needs to be calculated for the value of D in the above formula.

The total number of documents and the average document length($avgdl$) can be pre-computed and its a part of the pre-processing step. For the Reuters news corpus the total number of files are 809788.

B. Mapper and Reducer Phase

Mapper input: Inverted index file storing all the terms

Reducer output: Terms and the postings list that match the query

The mapper phase is same as for tf -idf, except that we pass the document length in addition to the document ID. In the reducer phase, we compute the weight for each document using the above formula. We have all the values we need to compute the weight using the above formula. We calculate the document weight for each query term and store it in a hash map. We have to perform the sorting of the document weights as before. The output of the reducer is thus the document with its BM25 weight.

A snippet from the inverted index file created for implementing BM25

```
msattir@user001:~$ cat fawn
fawn idf: 9.58128304605 17559newsML : 0.00229357798165 - 1.68737562832
q4jun96 idf: 9.58128304605 35961newsML : 0.0101351351351 - 1.14555
herve idf: 6.5855077249 35264newsML : 0.00243902439024 - 1.58675231103
35252newsML : 0.00163666121113 - 2.3646479562, 100208newsML : 0.00204081632
653 - 1.80936251806, 101415newsML : 0.00456621004566 - 0.847557941747, 130756
newsML : 0.00190839694656 - 2.02794685605, 35323newsML : 0.00404858299595 - 0
.955921514208, 36372newsML : 0.00280898876404 - 1.37776542129, 177055newsML :
0.00157232704403 - 2.4614011459, 35003newsML : 0.00142045454545 - 2.72456982
187, 177375newsML : 0.00189393939394 - 2.0434273664, 138143newsML : 0.006134
96932515 - 0.630830796825, 100831newsML : 0.00787401574803 - 0.491506203661,
34959newsML : 0.00180505415162 - 2.14405068369, 36108newsML : 0.0016556291390
7 - 2.33755706308, 35269newsML : 0.00165562913907 - 2.33755706308, 175508news
ML : 0.00280898876404 - 1.37776542129, 100208newsML : 0.00156494522692 - 2.47
301152806, 36183newsML : 0.002 - 1.93506379394, 138788newsML : 0.00181818181
818 - 2.12057017334, 138748newsML : 0.00694444444444 - 0.557298372656,
gat idf: 9.58128304605 99340newsML : 0.00221729490022 - 3.49085508427
publicize idf: 9.58128304605 3201newsML : 0.00316455696203 - 1.22296
031777,
ratning idf: 8.48267075738 99522newsML : 0.00307692307692 - 1.2577
9146606, 100275newsML : 0.00132802124834 - 2.91420607368, 98983newsML : 0.00
```

Fig. 7. Okapi BM25 Index File

VI. RESULTS

The comparison of the search techniques discussed in this report is discussed below.

A. Comparing Boolean and Ranked retrieval

- The Boolean retrieval can be useful only when the user knows the terms of the documents they want to search for. And the query narrows down the documents to a very small number. In all other cases, ranked querying is always better. Because the Boolean retrieval can also produce many results, we dont have a measure of the documents on how relevant to the query the documents are.
- All the search engines today perform a ranked query by default. But some also have options for Boolean retrieval. So between the Boolean and the two ranked retrieval, the ranked retrieval is always better.
- For the ranked retrieval methods, we have a list of documents that are ordered based on some parameters that can help the user as much as possible. This is not possible in Boolean query.

B. Comparing BM25 and tf-idf

- There are scenarios where one of the methods is better than the other. We found that for the Reuters news data, the tf -idf and BM25 performs more or less same for short queries. But for longer queries, we see that BM25 performs much better.

- For the Reuters news corpus, the document lengths are almost similar. So in the formula for BM25, we see that the L/L_{avg} also becomes a constant. Thus the value of b also doesn't affect the tf weight for that document. Only the value of k matters. As we have previously mentioned, if the tf is to be given more importance, then the k value should be higher and vice versa. On the other hand, tf -idf method, the tf weight grows logarithmically which means that more importance is given to the term frequency as compared to BM25. This is good for shorter documents, but for longer ones the tf weight should not be taken that high.
- BM25 performs very well for querying a library of book/novels. Since the novels do vary in length and BM25 takes into account the lengths of documents, it produces better results and that are relevant to the query. Tf -idf doesn't work that well with this type of data because tf weight is given more importance. But in novels few key words may be more helpful than other frequently occurring words.

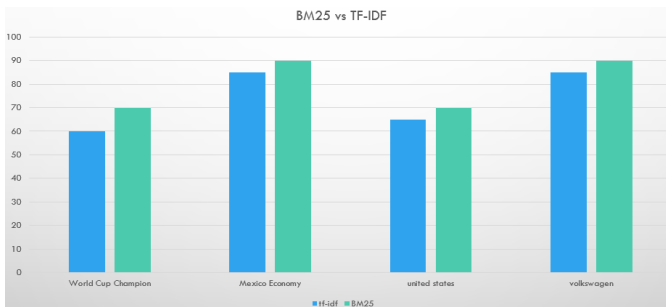


Fig. 8. TF-IDF vs BM25

VII. WEB INTERFACE

Creating a web interface for querying is one of the goals of this project. We have used JavaScript primarily to implement the interface. On the front-end, the website was created using HTML and JavaScript. On the back-end, we are using node.js to create the server and respond to the queries.

Steps in setting up the server:

- Create a job with the desired amount of resources that you need and login to the Name node.
- Create a secure tunnel from the users end to the servers location (ie) the Name node where the job needs to run. We chose SOCKS tunnel to achieve this.
- Create a server on the Name node. The server should respond with a website when its accessed from the users end. It should also be able to run the job in Hadoop if a query is requested.
- Once the query is received by the Name node server, we run the Hadoop job on this query and retrieve the output. This is then sent back to the users end and the output is rendered using JavaScript and CSS

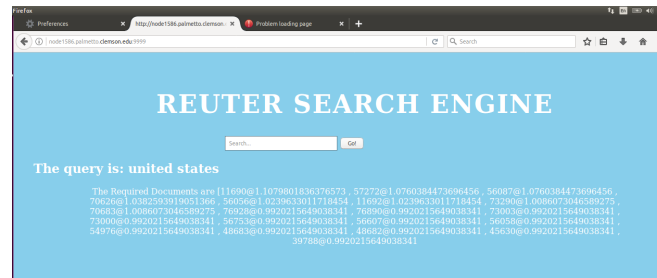


Fig. 9. Web Interface for the IR System

REFERENCES

- [1] Wikipedia
- [2] Course CPSC 8470, Spring '17, Lecture Slides
- [3] Stanford NLP Course pages